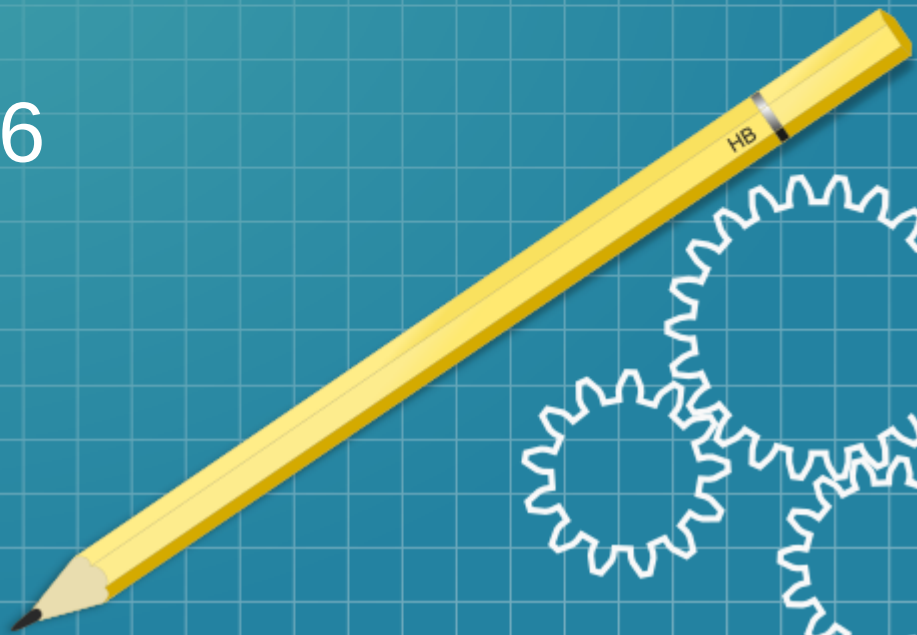
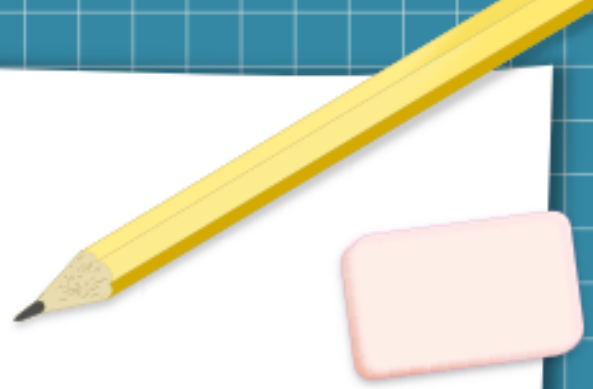
# Functions and an Introduction to Recursion

Chapter 6



# We will discuss:



- References and reference parameters
- Default arguments
- Unary scope resolution
- Function overloading
- Function templates

# References and Reference Parameters

There are two ways to pass parameters/arguments to functions in C++:

- *pass-by-value*
- *pass-by-reference*

# References and Reference Parameters

There are two ways to pass parameters/arguments to functions in C++:

- *pass-by-value*
- *pass-by-reference*

## *pass-by-value*

When an argument is passed-by-value, a copy of the argument's value is made and passed to the called function.

Changes to the copy *do not affect* the original variable's value.

# References and Reference Parameters

There are two ways to pass parameters/arguments to functions in C++:

- *pass-by-value*
- *pass-by-reference*

*pass-by-reference*

When an argument is passed-by-reference, the caller gives the called function the ability to *access the caller's data directly*, and to *modify that data*.

# References and Reference Parameters

A *reference parameter* is an *alias* for its corresponding argument in a function call.

# References and Reference Parameters

A *reference parameter* is an *alias* for its corresponding argument in a function call.

```
double area(double& l, double& w){

   return l * w;

}

double length = 10, width = 3.4;
area(length,width);
```

See the example in refParameters.cpp

# References and Reference Parameters

A *reference parameter* is an *alias* for its corresponding argument in a function call.

```
double area(const double& l, const double& w){

   return l * w;

}

double length = 10, width = 3.4;
area(length,width);
```

See the example in refParameters.cpp

# References and Reference Parameters

*Pass-by-reference* pros:

- when passing large objects, passing-by-reference avoids the overhead of passing a copy of the large object (const parameter can be used to secure the value of the original variable)

- if more than one value needs to be returned, passing-by-reference allows to ease that limitation.

# References and Reference Parameters

## Returning a *reference* to a local variable

A *reference* to a *local variable* can be returned:

```
double& area(const double& l, const double& w){

   double area;
   area = l*w;

   return area;
}
```

However, this can be dangerous, unless that variable is declared as static, since the local variable will be descarded when the function terminates.
An attempt to access such a variable yields *undefined behavior*.
Such references are called *dangling references*.

# References and Reference Parameters

## *References* as aliases within a function

*References* can be used as aliases for other variables within a function:

```
bool func(int a, const double& y){

   int b{5};
   int& c{b}; // creates c as an alias for b

   ++c; // increments b to 6 using its alias
}
```

Reference variables ***must me initialized*** in their declarations and ***cannot be reassigned*** as aliases of other variables.

# Default Parameters / Arguments

*Default parameters* are the rightmost arguments in a function's parameter list.

```
bool func(int x, double y, char x = 'y' ){

   ...
}
```

When using prototypes, the default parameter's value is given only in the prototype.

See defaultParameters.cpp

# Unary Scope Resolution Operator

C++ provides the *unary scope resolution operator* (::) to access a global variable when a local variable of the same name is in scope.

# Unary Scope Resolution Operator

C++ provides the *unary scope resolution operator* (::) to access a global variable when a local variable of the same name is in scope.

It cannot be used to access a local  variable in an outer block.

A global variable can be accessed directly, without the unary scope resolution operator, if the name of the global variable is not the same as a local variable's name.

# Unary Scope Resolution Operator

C++ provides the *unary scope resolution operator* (::) to access a global variable when a local variable of the same name is in scope.

It cannot be used to access a local variable in an outer block.

A global variable can be accessed directly, without the unary scope resolution operator, if the name of the global variable is not the same as a local variable's name.

see unaryScope.cpp

Recommendation: always use the *unary scope resolution operator* to refer to global variables, even if there is no collision with a local variable's name. It makes programs more readable, easy to modify and avoids unexpected local variables that hide the global variable.

# Function Overloading

C++ allows several functions of the same name to be defined, as long as they have different signatures.
This is called *function overloading*.

See the example in functionOverloading.cpp

# Function Templates

Python doesn't associate types with variable names, so the same code might work for different types.

The function Maximum finds the larger of two numbers having the same type (as long as the operator > is defined for that type).
For example, the types *int*, *float*, and even *string* will work here:

```
def Maximum(a, b):
   if a > b:
      return a
   else:
      return b
```

Dynamic typing is possible in Python because the *interpreter* waits until it is ready to execute a Python statement before converting it to machine language.

# Function Templates

In C++ we have learned that C++ variables must be defined with a fixed type, so that the compiler can generate the specific machine instructions needed to manipulate the variables.

```
int maximum_int(int a, int b) {

   if (a > b){
      return a;
   }

   else {
      return b;
   }
}
```

# Function Templates

In C++ we have learned that C++ variables must be defined with a fixed type, so that the compiler can generate the specific machine instructions needed to manipulate the variables.

```
double maximum_double(double a, double b) {

   if (a > b){
      return a;
   }

   else {
      return b;
   }
}
```

# Function Templates

*Template* mechanism in C++ that allows to write functions and classes with similar to Python's functionalities.

C++ templates allow us to write one version of the code, and the compiler automatically generates different versions of the code to each data type *as needed*.

```
template <typename T> // or template <class T>
T maximum(T a, T b) {

   if (a > b){
      return a;
   }

   else {
      return b;
   }
}
```

Comment: you may use any legal identifier instead of T. The other commonly are Item and Type .

see templateFunctionExample.cpp

# Function Templates

- The C++ compiler doesn’t generate any code if no template function is called

- Depending on compiler, it may or may not catch syntax errors in template functions that are not called, hence it is important to test all the template functions

- The term instantiate is used to indicate that the compiler generates the code for a specific type.

# HW assignment

1) Exercise 6.49
2) Exercise 6.51

Suggested exercises
(*not for grade, but the questions related to these will appear on a quiz or a test*):
1) Chapter 6,
   Summary and all Self-Review Exercises (pages 203-205)
2) Chapter 6, Exercises: 6.46, 6.53